NASA Contractor Report 4244

Predeployment Validation of Fault-Tolerant Systems Through Software-Implemented Fault Insertion

Edward W. Czeck, Daniel P. Siewiorek,
and Zary Z. Segall

GRANT NAG1-190
JULY 1989

NASA Contractor Report 4244

Predeployment Validation of Fault-Tolerant Systems Through Software-Implemented Fault Insertion

Edward W. Czeck, Daniel P. Siewiorek,
and Zary Z. Segall
Carnegie-Mellon University
Pittsburgh, Pennsylvania

Prepared for
Langley Research Center
under Grant NAG1-190

NASA
National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Division

1989

Contents

1 Introduction **1**

2 Validation **2**

2.1 System Specifications 2

2.2 Validation Methodologies 3

3 Faults **5**

3.1 Origin of Faults 5

3.2 Fault Models 6

3.3 Fault Insertion Examples 8

3.4 Other Fault Insertion Examples 11

4 FIAT **12**

5 The FIAT Process and Its Abstractions **17**

5.1 Workload Management 17

5.2 Fault Classes 20

5.3 Experiments 22

5.4 Data Collection and Analysis 24

6 Implementation **25**

6.1 Hardware 25

6.2 Software 25

6.2.1 FIM Software 26

6.2.2 FIRE Software 27

6.3 Experimental Procedure 30

7 Evaluation **31**

7.1 Experiment 1 31

7.2 Experiment 2 34

8 Summary **37**

9 References **38**

1 Introduction

Validation methodologies of ultra-reliable systems have been explored by NASA[47,48]. The developed methodologies included fault-free characterization of the system, single processor behavior under faulted conditions, and system behavior under faulted conditions. CMU's efforts have supported this validation methodology through empirical studies. Early work by Clune, Feather, Czeck, and Grizzaffi[12,15,16,22] determined the value of fault-free characterization. Additionally, Feather[22] developed a synthetic workload generator and monitor to aid in the characterization of the system, while Czeck[16] supplemented the workload generator to induce faults into the system. Further work at CMU lead to the inception of the FIAT project.

This paper describes FIAT, a Fault Injection-based Automated Testing environment, which is used to experimentally characterize and evaluate distributed realtime systems under fault-free and faulted conditions. Section 2 surveys validation methodologies and develops the need for fault insertion. Section 3 overviews fault origins and fault models. Section 4 suggests some motivations for FIAT and presents a high level description of the system. Section 5 discusses the abstractions used in the FIAT system, while Section 6 details its implementation. Finally Section 7 describes two experiments performed on the FIAT system, and Section 8 concludes with a summary and observations.

2 Validation

Validation is the process of substantiating, through demonstration, that a given system meets its specification[4,5]. For highly dependable systems, the specifications contain extreme reliability requirements which necessitate the ability to function under faulty conditions. To demonstrate or validate the system operation, prediction methods must be used to determine the system "operation point" before the system is committed to use. Methods of determining the "operation point", or the nominal behavior, include simulation, modeling, and analysis. Complimentary to these methods are experimental methods, such as fault insertion, which are well suited for areas in which modeling and analysis fail to capture the needed detail.

2.1 System Specifications

System specifications can be divided into two domains, with the validation effort directed to demonstrate that both are fulfilled[11]. The first domain includes functionality and the second is the bounds within which correct functioning must occur. Functionality is by far the easier of the two domains to validate; metrics, such as throughput and realtime deadlines, are readily defined. The bounds of correct functioning typically are associated with dependable computing and include metrics such as reliability, maintainability, and fault tolerance. Although the two domains cannot truly be separated during validation, this report shall concentrate on the validation regarding the reliability and fault-tolerance issues. Previous reports outlined fault-free validation[12,15,16,22].

A typical reliability requirement for a life critical application is 10^{-10} failures per hour. The basis for this failure rate can be justified through the following life-cycle model. Assume a 30 year life, with 8 hours operation per day; this yields approximately 100,000 (10^5) operational hours per unit. If 100,000 (10^5) copies are produced and one failure is acceptable over the life of all copies, then the failure rate must be less than 10^{-10} failures per hour. This translates to one failure per 1 million years per unit or several orders of magnitude greater than the reliability of todays systems. This stringent reliability requirement yields two observations. The first is that non-redundant systems are at least six orders of magnitude less reliable than the goal, necessitating the use of redundancy and its ability to function correctly with faults present. The second is that life testing (monitoring) for confirmation of reliability is impossible, necessitating the need for accelerated testing. These observations formulate the problem addressed by this report: how can the reliability specifications be validated.

The need for a validation environment, such as FIAT (Fault Injection-based Automated Testing environment) which is described in this report, stems from system specifications. Two possible methods exist for validation of high reliability systems under faulted conditions. The first method, life testing, monitors actual running systems awaiting the natural occurrence of faults. The behavior of the system, when faults occur, can then be analyzed and used to support validation assumptions or conditions. The second method, fault insertion, induces faults into the system and analyzes behavior under these conditions.

Life testing offers realism, but due to the current level of component reliability, faults can be expected at a rate of one in 10^3 hours per system. This failure rate is prohibitively slow for the completeness required in thorough testing. Fault insertion speeds up the rate at which synthetic faults occur. The use of synthetic faults is necessary given the large number of fault types, fault locations, and times of occurrence. For example, a small board consisting of 50 packages each with 20 pins, has a fault space of 1000 pin-level faults without considering any time dependencies. Additionally, software faults must be considered as the majority of system complexity moves into software. The software-fault space is also large: consider the amount of code present in even the smallest of operating systems.

2.2 Validation Methodologies

Much work has been done in validation methodologies, especially in aerospace and other life-critical applications. These methodologies include formal proofs, analyses, and tests to assure the system meets its specifications. Although there is no commonly accepted validation methodology, a generalized methodology may be extracted from procedures presented in the literature[11,20,27,29,45,47,48,51]. The approach is to build confidence in the system by a thorough and systematic methodology of proofs, analyses, and tests. Proofs are formal arguments supported by deductive inferences. Analyses employ models of the system, and testing uses statistical inference. These three methods are complementary: proofs and analyses use abstract models of the systems; testing uses the actual system to substantiate the models and results generated in the analysis.

These three processes (proofs, analyses, and testing) are applied throughout the system development as depicted in Figure 1. During the architecture development, proofs are generated which specify the conditions necessary to achieve the requirements. Analysis of the architecture include reliability and error rate Markov models, while the testing comprises activities such as high level simulations and design reviews. At the implementation level, the conditions required in the architecture proofs are verified, leading to more conditions for the realization. The analyses include further refinement of the Markov models developed in the architecture analysis, and in-depth analyses such as Fault-Tree generation. Testing begins to involve concrete methods such as simulation and emulation of the design.

In the final level realization, proofs of the hardware and software structure are continued from the implementation level. Analyses include exhaustive Failure Modes and Effects Analysis, refinement of Fault-Tree analysis, and the inclusion of specific failure rates into the reliability and error rate analyses. Testing at the realization stage measures the assumptions and requirements used in the proofs and analyses. The assumptions involve error rates, fault latency, and coverage, as well as concrete measurements such as throughput, utilization, and error recovery time.

Development Level	Abstract		Concrete
	Design Proofs	Analyses	Tests
Architecture	Prove Architecture Against Requirements	Reliability and Error Rate Markov Models	Design Reviews High Level Simulations
Implementation	Prove Implementation Against Architecture	Fault Tree Analysis	Simulation and Emulation of Hardware
Realization	Prove Realization Against Implementation	Failure Modes and Effects Analysis	Support Assumptions from Analysis, Fault Insertion

Figure 1: **Validation Areas Throughout the Design Process**

3 Faults

The validation of dependable systems requires an understanding of faults. This section discusses faults, their origin, models of faults, their basis, and some fault insertion examples.

3.1 Origin of Faults

One possible classification of faults is by their origin. Three categories arise: design errors, manufacturing introduced faults, and faults occurring in field use. The following paragraphs discuss these three fault origins[1].

Design errors are caused by improper translation of an idea or concept into an operational realization[56]. Gathering information on design errors is difficult because each design error occurs only once per system and the types of design errors are usually complex. Design errors have been studied mostly in the software realm due to the expanding requirement of reliable software and the fact that all software failures reduce to design mistakes. The effects of these errors are difficult to generalize due to their diversity and infrequency of occurrence, hence few if any models have been developed to characterize their behavior.

Manufacturing defects are introduced by improper processing or the use of defective material in the fabrication of the system. Examples of manufacturing defects in semiconductors include: cracked dies, over or under doping of material, and flaws in the mask. The effects of manufacturing defects on CMOS circuit behavior have been characterized in Ferguson[23]. Ferguson applied mask defects to several industry circuits and extracted the behavior from the faulted circuit. He noted from the results that over 99% of all faults can be characterized as either bridging faults or breaks, while only approximately 50% of all faults can be modeled as single stuck line faults.

Field failures are caused by physical processes occurring through normal and abnormal use during the life of the system. Examples of MOS device failures are listed below[21,39,46].

1. Thin oxide breakdown is a primary mechanism which is due to large electric fields in the insulator, usually in the gate oxide.
2. Electromigration, the drifting of metal atoms toward the cathode, is a common wear out mechanism influenced by high current densities in the conductor.
3. "Hot electron" trapping in the gate oxide is caused by high temperature and high electric fields in the channel of a MOS transistor.
4. Soft or transients errors are produced by alpha-particles and cosmic radiation which create several electron-hole pairs affecting stored charges.
5. Electric overstress due to improper environmental conditions such a electro-static discharge, may cause multiple physical failures.
6. Other life failures are caused by an array of sources: ranging from design deficiencies, production techniques, mechanical stress, corrosion, etc..

Little has been published on the distribution of field failures due to the variation introduced by the manufacturing process and operation environment. Additionally, knowledge regarding the effects of field failures on circuit behavior is limited, but several sources give isolated information. Lloyd and Knight[38] empirically support the classification of shorts and opens due to electromigration into a single failure model, but give no information on the resulting behavior. Timoc[63] attempted to map physical failures to logical models; several failures resulted in "stuck-at" behavior, while others resulted in parametric faults.

Fault Effect:	Manufacturing Stage	Field
Oxide	less than 10%	25% - 75%
Metal	30% - 40%	4% - 17%

Figure 2: **Fault Effects: Manufacturing Stage vs. Field Life**

It is interesting to contrast faults originating in manufacturing and field operation. In field operation, the frequency of failures is highest for gate oxide, and next highest for opens and shorts in metal runs (see Figure 2). Oxide failures result in transistors stuck off (for both oxide breakdown and electron trapping). Metal failures result in open and shorted metal lines, breaks and bridges (for electromigration). During manufacturing on the other hand, Ferguson[23] reports less than 10% of mask defects cause oxide problems and 30% to 40% cause extra or missing metal, with an insignificant percent of faults causing transistor stuck off faults.

3.2 Fault Models

Fault models are abstractions of failure mechanisms to a level of understanding desired by the user. Models range from the physical device level to the gate, functional and even architectural level. A wide range of models have been developed for test program generation. Figure 3 summarizes fault models at different levels of abstraction. Fault models are typically categorized according to the origin of the defects: manufacturing or field operation. Most fault models are based on manufacturing defects, which is only of concern during a fraction of the system lifetime. Validation is interested in the events occurring during the useful life of the system, hence some of the manufacturing fault models are not applicable.

Switch-level fault models[9] are used for MOS devices where unidirectional logic gate models do not adequately detail the bidirectional behavior of such devices under certain fault types (e.g., bridging, stuck open and closed transistors.) Switch-level models contain nodes, connected by bi-directional transistors (switches); faults are nodes stuck high or low, transistors stuck open or closed, and extra or missing transistors. Low level simulations and models are required because the failure modes of devices, especially CMOS[25], cannot be modeled at the gate or higher levels. Furthermore, limited access to individual devices prohibits empirical observation. Switch-level simulations aid in test program generation and test coverage assessment, but the simulation cannot be used for large systems as modeling and simulation time become prohibitive.

Gate-level fault models assume inputs and outputs of gates are stuck at high or low logic values, but the gate functions correctly. These faults are based on printed circuit (PC) boards, TTL, and pre-TTL

Level	Fault Models	Basis	Limitations
Network	Communication lost, delayed or unordered. Lost nodes.	Abstraction of behavior.	Failure modes are unknown and complex.
PMS	Data change. Message or process lost. Data inconsistent. Time outs.	Abstraction of behavior.	Failure modes are unknown and complex.
RT	Data change. Wrong assertion, source, or destination.	Abstraction of behavior.	Based on RT models not implementation.
Functional	Complement or dual function Truth table modification.	Observation? Ad hoc.	Many realizations and fault modes.
Gate	Gate output stuck at 0 or 1, Single stuck line.	TTL and PC board behavior.	Technology outdated.
Switch	New and missing devices, Shorts and breaks (opens), Transistors stuck on/off.	Processing defects.	Simulation overhead, difficult to observe and fault insert.

Figure 3: **Fault Models at Different Abstraction Levels**

logic: circa 1960. The gate-level fault model is not applicable to MOS implementations as failure modes are possible which transform a combinational MOS circuit to a sequential circuit[64][1] and complex MOS circuit implementations do not map gate lines to circuit nodes. Beh et al[7] emphasized that fault models should be consistent with manufacturing defects and developed a methodology relating TTL processing defects to their logic behavior. Beh's work was limited to demonstrating which defects can be modeled by gate-level stuck-at faults and did not attempt to develop new fault models.

Ferguson et al[23,54] developed fault models based on processing mask defects and mapped the defects to gate-level fault models, illustrating that only 50% of the faults are representable by a gate-level single stuck line fault. Marchal[40] challenged the stuck-at models used in functional testing with simulations of faulted microprocessor internal buses. Results show fault models should be realization and technology dependent, with models updated as technology advances. These approaches require realization details and do not further abstract faults to higher levels.

As integration levels increase further, testing based on implementation faults becomes prohibitive and functional testing approaches, either implementation dependent or independent, must be used[28,59]. Thatte and Abraham[60,61,62] presented the ground work for functional testing, with fault models based on possible scenarios of failures occurring within the control section, data section, or data storage of a generalized microprocessor. These models were implementation independent, hence they did not embody a complete fault library, but they were used to generate test procedures for microprocessors. Silberman and Spillinger[57] present a formalized methodology to define the functional-level fault model, given the implementation of a circuit, its related defects, and the input vector. The resultant functional fault model is then used in simulations to acquire a better estimate of implementation fault coverage

[1]Results from Ferguson[23] show the occurrence of these types of faults from manufacturing defects to be less than 2% of all faults occurring.

saving the overhead of low level simulation.

Architectural-level testing has been proposed and implemented[17,18], with mostly ad hoc fault models. Stuck-at faults of input and output nets, mutation operation in the microprocessor (Thatte and Abraham model) and in control resources (instruction fetch and decode units) have been included in the architectural-level fault model. The evaluation of these models was based on test coverage results from simulations at the architectural and gate level, with results of the architectural simulations, at best, tracking the coverage of the test programs, and showing which fault models are appropriate.

Throughout the referenced papers, the goal was to substantiate or formalize the fault models used at the higher levels based on known failure modes[2] or lower level fault models. There are still some limitations in predicting higher (component) level fault behavior, even with fault models based on manufacturing defect distributions. Using a gate-level emulation of an avionic processor, McGough et al[41,42,44] showed an 87% coverage of all gate-level faults contrasted with a 98% coverage for all component[3] (pin) level faults. Furthermore these studies concentrated on logic value faults and did not consider parametric or other faults.

Some results from software testing support the concept of functional testing based on abstract fault models. One interesting result is the comparison between "black-box" testing, similar to the functional-level in hardware, and structural testing, similar to the component level. In "black-box" testing the internal structure of the module is unknown. The test input generation proceeds from specifications given to the module, with emphasis on boundary values of both the input and output vectors. In structural or "white-box" testing, the internal structure of the program is known, and the test generation typically attempts to exercise *all paths*. In a study by Howden[30], "black-box" testing was significantly more effective in error detection than structural testing due to subtleties in the data selection. Data for "white-box" testing are geared to exercise each path once for some data, but data for "black-box" testing are geared to exercise the program for the range of valid data. Also Lai in his Ph.D thesis[32] and in two conference papers[33,34] showed the merits of functional testing of hardware without the details of hardware implementation.

3.3 Fault Insertion Examples

The evaluation of a fault-tolerant systems includes both functional and reliability measures. Functional measures are relatively straight forward, but the analyses of the reliability requires measures of such quantities as component reliability, fault coverage, error detection coverage and other difficult to measure factors. These unique measures typically require special methods, such as fault insertion[11,47,48,51].

Fault insertion has been used extensively for a number of objectives: test coverage evaluation[31], the generation of fault dictionaries[26], the study of error propagation and latency[13,24,44,55], the evaluation of error detection schemes[14,52,53], and system evaluation[6,36,58]. Figure 4 summarizes these studies. By far, the two most common means of fault insertion has been simulation of the hardware

[2] Most of the failure modes considered were during the manufacturing phase.

[3] A component, as used in this study, is defined as a single SSI, MSI or LSI chip.

Source	Target	Method/Level	Goal
Avizienis '72 [6]	JPL Star breadboard	Permanent and Transient/ Pin (Gate) Level	Estimate of Detection, Recovery Procedures, and Coverage
Goetz '72 [26]	ESS Microstore	Simulation/ Gate Level	Detection and Coverage Measure
Courtois '79 [13]	MC 6800	Simulation/ op-code (RT) level	Detection Time
Decouty et al '80[19], '82[14]	GORDINI: Fault Tolerant micro	Physical/ Pin (Gate and RT) Level	Tool and Methodology Development
Kurlak '81 [31]	GE MCP-701 CPU	Physical Faults with FMEA analysis	Evaluation of Watchdog Timer
McGough & Swern '81 [43]	Bendix Simplex BDX-930	Simulation/ Gate and Pin Level	Coverage Measurement
Lala '83 [36]	FTMP Engineering Model	Physical/ Pin (Gate) Level	System and Coverage Evaluation
Yang et al '85[65]	iAPX 432	Fault Emulation/ Memory Words (RT)	Coverage of TMR
Schuette et al '86[53]	MC 68000	Transient, Physical/ Bus (RT) Level	Evaluation of Error Detection Techniques
Czeck et al '87[16]	FTMP Engineering Model	Fault Emulation/ RT Level	Methodology Study
Finelli '87 [24]	FTMP Engineering Model	Permanent Physical/ Gate Level	Fault Recovery Distributions

Figure 4: **Fault Insertion Examples**

and physical fault insertion. Fault simulation has occurred at all the levels discussed in Section 3.2 on fault models[9,43,49,57]. Physical fault insertion has been used to determine fault coverage of test programs[6], fault latency[24,55], and fault detection efficiency[14,36,53].

Figure 5 presents a matrix of advantages and disadvantages of fault insertion as a function of methods. The four methods of fault insertion include:

1. Software simulation involves fault insertion by code modifications or special functions of the simulation engine.
2. Hardware emulation uses hardware representative of the system under test, such as an engineering prototype, as a basis for study.
3. Fault emulation attempts to imitate fault behavior through software control of the hardware or special capabilities built into the hardware.
4. Physical fault insertion involves the inducement of faults through special hardware to the actual system under test.

	Fault Insertion Methods			
	Software Simulation	Hardware Emulation (Breadboard)	Fault Emulation	Physical Fault Insertion
Advantages	Access to system at any level of detail. Fault types and control are unlimited.	Representative hardware with favorable access and monitoring.	True hardware and software in use.	True hardware and software in use.
Disadvantages	Simulation time explosion. Lack of tools limit ability.	Implementation and other parameters will change with deployed system.	Fault types are limited.	VLSI limits access and monitoring points. Task is difficult.

Figure 5: **Fault Insertion Methods**

Physical fault insertion has been used extensively in system validation, with the typical means of fault insertion being pin-level stuck-at and inverted faults[6,14,19,36,53,55,58]. With a SSI/MSI realization of a system, pin-level stuck-at's closely represent failures which have been observed to occur in such devices; but with LSI and VLSI realizations, failures may be remote from the input/output pins. At these higher levels of integration, fault insertion seldom claims to accurately portray physical faults, but the hope is that they provide a first approximation to the metrics under study. Palumbo[50] hypothesised that pin-level stuck-at faults produce error behavior similar to error behavior caused by internal devices. Initial empirical results are inconclusive; the hypothesis holds well for 85% of the data, but other data call for rejection.

Although actual faults may be remote from the pin boundaries, promising results have been reported with pin fault insertion of LSI and VLSI devices, and other abstract fault insertion methods. Schuette et al[53] inserted transient faults on the data, address, and control lines of an MC68000 bus, representing faults within the data and control sections of the processor. With this fault insertion ability, two error-detection schemes were evaluated. Yang et al[65] inserted faults into an iAPX 432 to evaluate software implemented TMR; the faults were generated by altering bits in the program or data areas in memory using the debugger. Czeck et al[16] inserted faults in an FTMP triad by causing one processor to execute special code, thus triggering the error detection mechanism. This method was able to duplicate some hardware fault insertion results presented by Lala[35,36]. But even with these results, McGough[41,42,44] illustrated a distinct gap between gate level and component level fault types as discussed in Section 3.2.

3.4 Other Fault Insertion Examples

Aside from hardware test generation and verification, fault insertion has been used in other areas. One area, which is related to computing systems, is software testing. Two software fault insertion examples are error seeding and mutation analysis[2,3,4,10]. While both methods seem similar, there are subtle differences. Error seeding is the process of inserting faults (errors) into the software during debugging. When the acceptable percentage of the the seeded errors are found, the debugging effort ceases. It is assumed that the unseeded errors (programmers errors) are found at the same rate as the seeded errors, thus error seeding provides a measure of the number of unfound errors. Mutation analysis is used as a measure of test program coverage. Faults (mutations), which are a slight perturbation of a program statement, are inserted one at a time into lines of code. Test sets are run to determine if the mutation is detected, thus gaining a measure of the test effectiveness.

Within the Sperry Univac 1100/60[8], fault insertion capabilities are built into the system to verify the functionality of the fault detection, isolation, and recovery mechanisms. Fault insertion is activated during system idle time and can insert faults in the processor, memory, and I/O unit. These fault insertion capabilities are under operating system control and require no external hardware.

4 FIAT

FIAT, Fault Injection-based Automated Testing environment, is a prototype of an experimental test bed used in the exploration of validation processes for fault-tolerant systems. System validation requires the ability to monitor the system under test, the ability to control the system to induce faults and other operating conditions, and the ability to repeat tests to identify the source of system deficiencies. The requirement for experiment repeatability, as well as the need to attack a large fault space, are the reasons FIAT provides a test environment capable of automatically inducing faults and monitoring system behavior.

The underlying methodology which guides the FIAT validation process is the following:

- Specify an architecture through models, emulation, or prototypes.
- Specify the system software architecture and workload.
- Profile the fault-free behavior of the system to determine a nominal "operation point".
- Select a set of fault classes for fault insertion experiments.
- Perform fault insertion experiments.
- Analyze experimental data and use the results to support validation requirements and compare fault-tolerance strategies.

Given this desired validation methodology, the goal of the FIAT project is to develop an environment for automated software fault/error insertion, error detection, and recovery analysis which can be used to perform a more thorough investigation of software implemented fault insertion and how software fails. To perform the desired validation methodology FIAT provides an environment capable of supporting the following functions; these are described in greater detail in Section 5.

Architecture Development: The development methodology for an architecture can be divided into three sections: simulation, emulation, and actual implementation. The hardware and communications structure of the FIAT system is general, so it may emulate a variety of architectures under evaluation. The ideal architecture for FIAT to emulate is a message-based, replicated structure, where messages are passed via the FIAT communication channels and the replicated structure is emulated by FIAT hardware and software tasks. This allows the user to design and evaluate a system without customized hardware or software, or a large initial effort. Examples of this flexibility can include systems such as: a Tandem like structure which employs a primary and secondary, a Stratus structure employing duplicate and compare, and a SIFT-like system employing several redundant processors and voting.

Moreover, the FIAT methodology is not limited to architecture emulation, as the FIAT environment can be transferred to the actual implementation. Benchmarks conducted in the emulation phase can be redone with the actual implementation to calibrate the system and further experiments developed to evaluate the implementation.

Additionally, FIAT provides several tools which aid in the development of the workload and hence the experimental architecture. These tools, available through the experiment interface or directly to the user, include editors, compilers, and the ability to provide monitoring "hooks" into the system.

Fault Insertion: Fault insertion must exercise the error detection and recovery mechanisms (EDRM) as well as develop quantitative metrics of the dependability properties of the system under test. Using software fault insertion, FIAT can insert faults or the manifestation of faults (i.e., errors) by either triggering error detection mechanisms or by seeding errors into memory. Software fault insertion was selected for the following reasons:

- Systems to be validated have a substantial software component. Software fault insertion allows penetration into the software portion of the system as well as exploring the interaction of software with hardware.
- Software fault insertion is less expensive, in terms of time and effort, than hardware fault insertion.
- Software fault insertion is functionally complementary to hardware fault insertion and does not exclude it.
- There is a need for a testing methodology to validate software implemented fault tolerant strategies.

Hardware (error insertion)	Operating System	User Application	System (OS and Application) (error insertion)
Bus	Code	Code	Lost, Delayed Messages
Memory	Data	Data	Corrupted Message
Registers		Registers	Delayed Task
I/O structure			Abnormal Task Termination
			Timer Corruption
			System Clock Corruption

Figure 6: **Faults and Errors Available in FIAT**

Software fault insertion has its limitations, mainly in its inability to force low level errors, such as a gate output stuck-at. However, designers are interested in the behavior of the whole system (hardware and software). Furthermore a large amount of the hardware functionality is visible through software.

Figure 6 is a table of the faults and errors available and currently under development for software insertion in FIAT (unless specifically noted, all of the insertions are faults). The use of *fault* and *error* in this figure and throughout this report refer to the definitions proposed by Laprie[37] and Siewiorek[56]. A *fault* is an erroneous state of hardware or software, whereas an *error* is the manifestation of a fault. The duration of inserted faults range from transient to pseudo-permanent[4].

Automation and Unity: The quality of fault insertion experimentation is a function of the capability of the system to insert (test) as many faults as possible per unit of time and of the fidelity of the fault insertion method itself. Automation includes both *experiment development time* and *experiment run time* processes.

[4]Pseudo-permanent is achieved by repeating a transient fault over a long duration. True permanent faults are difficult to emulate because software tends to overwrite storage locations.

To be effective, the various components of the system (e.g., workloads, fault classes, experiments and data analysis) must be integrated under one comprehensive environment, which supports the process of preparation, debugging, run time control and data analysis. Each of these components are discussed in the following section.

The CMU hardware implementation of FIAT uses four IBM RT PCs, connected via a 10 Mbit token ring, Figure 7. The hardware provides the execution platform for FIAT, without limiting the generality of the system. Hence a myriad of architectures can be emulated through software without any limitations imposed by the hardware or communications structure. The implementation is not limited to the RT and can be ported to other systems.

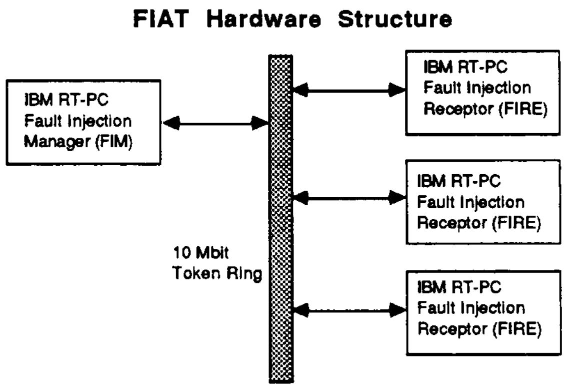
FIAT Hardware Structure
IBM RT-PC Fault Injection Manager (FIM)
IBM RT-PC Fault Injection Receptor (FIRE)
IBM RT-PC Fault Injection Receptor (FIRE)
IBM RT-PC Fault Injection Receptor (FIRE)
10 Mbit Token Ring

Figure 7: **FIAT Hardware Structure**

The FIAT software structure is more complex than the hardware structure and is divided into two parts: the Fault Injection Manager (FIM) and the Fire Injection Receptor (FIRE). Figure 8(a) shows the FIM software consisting mostly of the experiment interface controlling all phases of the experiment, from design to data analysis. Several intermediate forms are created in the process; the user defined workloads, the fault lists, and experiment descriptions are created and compiled from users specifications and FIAT libraries.

Figure 8(b) shows the software configuration on the FIREs. The core element is the fire monitor and control (FMC), which interfaces the fault list and the experiment description to the workload. The FMC also monitors the execution of the experiment sending appropriate information to the data files. Further details of the hardware and software are presented in Section 5 and Section 6.

As an example of the methodology's versatility, Figure 9 shows a possible emulation of the SIFT system using FIAT. The processors are emulated via software, with two SIFT processors running per FIRE machine. The scheduler along with SIFT tasks, such as voting, synchronization, sensor input, and applications, are run under the virtual SIFT processor. Communications channels emulate the point to point communications of SIFT with a dedicated channel per physical SIFT link. Using FIAT with

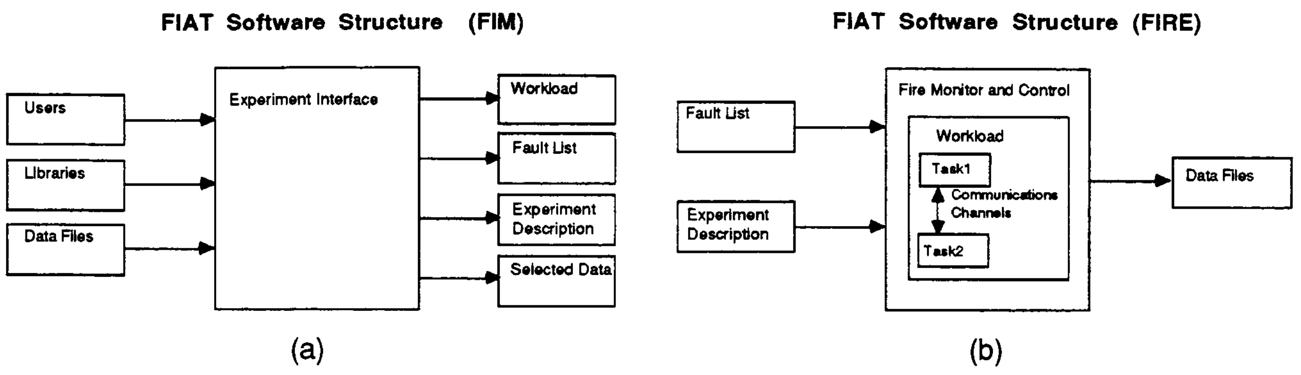

Figure 8: **FIAT Software Structure**

this application, several experiments may be run, including validation of the voting and interactivity consistency algorithms using fault insertion, measurement of task overhead, and others.

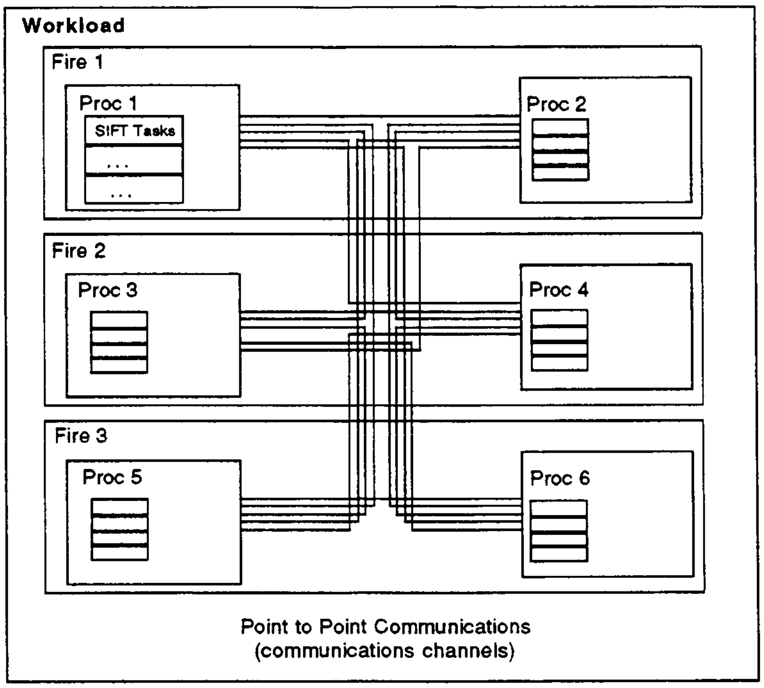

Figure 9: **SIFT Emulations on FIAT**

5 The FIAT Process and Its Abstractions

Following the methodology defined in Sections 2 and 3, an experimental validation using FIAT may progress through stages:

- Fault-free validation of a system: profile the system components' performance characteristics and collect data to determine the system's "operating point".
- Fault-free validation of a workload: profile the performance/functionality characteristics of the workload and collect data.
- Fault insertion experimentation: insert faults into profiled workload, then collect histories, and error records.

To support the fault insertion environment, FIAT provides the abstractions presented in Figure 10, each described in the following subsections. FIAT also provides the necessary tools to operate on the abstractions during experiment development and runtime.

	Workload Management	Fault Class	Experiments	Data Collection and Analysis
Abstractions	Task Attributes, Task Image	Fault Class, Fault Instance	Experiment Definition and Script	Histories, Error Records, and Reports
Operation	Edit, Generate Task Image	Edit, Generate, Fault Instance	Edit, Generate Experiment Script	Instrumentation, Database
Runtime	Monitor, Control	Fault Insertion	Script Execution, Script Interpretor,	Data Collection, Data Analysis

Figure 10: **FIAT Abstractions**

5.1 Workload Management

The workload of a computer is defined as the set of all inputs (programs, data, commands) the system receives from its environment. A workload can be classified as *natural* or *synthetic*. Natural workloads accomplish useful work, while synthetic workloads model natural workloads. The advantage of using a synthetic workload to profile a system has been demonstrated by previous CMU work[15,16,22]; hence FIAT manipulates synthetic *workloads*(WL). In FIAT, a workload is an *observable* set of real-time communicating tasks representing the user's real-time programs.

A task is an observable (monitorable) unit of computation, defined by the user, within the workload, which communicates through observable communication media named *channels*. Note that a workload could run in one machine or be distributed among a number of computers. Figure 11 shows an example, in C language source code, of the matrix multiplication task. The `create_channel` construct establishes

```
task(argc,argv)
int argc;
char **argv;
{
    create_channel("mat1");
    sensor(11);
    compute();
    sensor(12);
      .     .     .
}

compute()
{
    for (i=0;i<3;i++) {
        for (j=0;j<3;j++) {
            for (k=0;k<3;k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

Figure 11: **Typical Task Definition**

a communication medium which allows other tasks to communicate with the task.[5] The `sensor` construct is a user-defined means of observing the execution of the task through executive function calls embedded in the workload tasks. Its two basic functions are to provide a mechanism to monitor the execution of the workload tasks and to synchronize communicating tasks.

For symbolic fault insertion, a number of task attributes must be extracted from the workload. The term attribute refers to the set of symbolic names identifying the tasks, the code, and the data segments within each task. This analysis is automatically performed by a tool known as the *attribute extractor*. The attribute extractor, after analyzing the workload, provides data constructs (e.g., task tables) known as *domains*. These domains are used for the automatic generation of fault lists. Domains include task and function names, variable names, and locations of code and data within the object file(s).

As depicted in Figure 12, each task is linked with four program attachments which respond to external requests. The *workload monitor* provides observability of execution through the sensors, the *fault injector* performs the fault insertion, the *error detection and reporting* attachment provides the data collection capability, and the *fault-tolerant architecture* is a generic capability for implementing software-based recovery mechanisms. Figure 13 illustrates the process of attribute extraction and the linking of the attachments in the generation of a task.

[5]Text in a `typewriter` font refer to program constructs or FIAT commands.

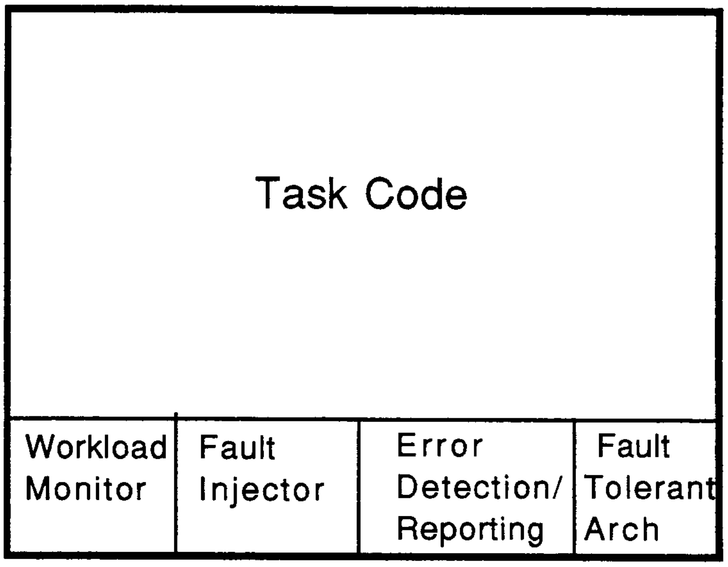

Figure 12: **Typical FIAT Task Modified by Attachments**

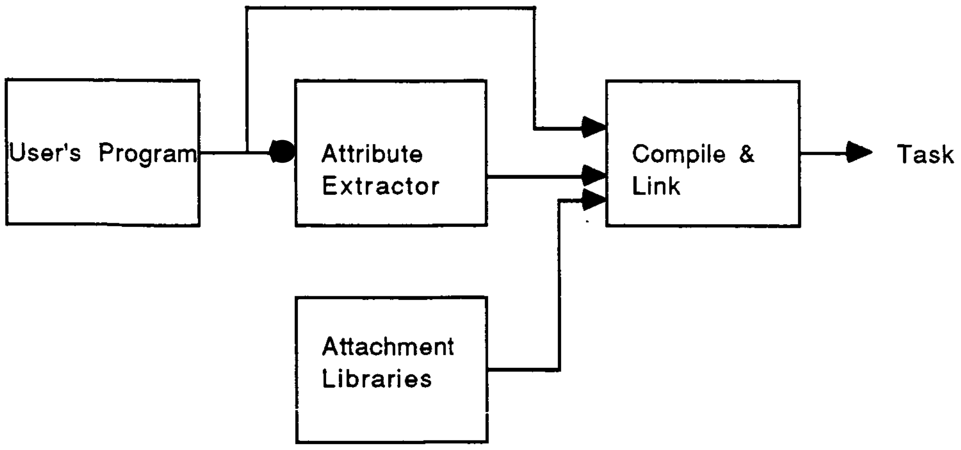

Figure 13: **Task Generation**

5.2 Fault Classes

A fault class is a template which describes a set of workloads or system modifications representing a group of physical or logical faults having common properties. This template is similar to an abstract data type. Figure 14 shows the fault class format, where field-i is the identifier of the Domain-i[6] and method-i is a user defined procedure for selecting items from the domain (Domain-i) to be used in fault insertion.

As an example, Figure 15 shows a memory fault class. The line, Mechanism: fil, shows the fault insertion method used. In this example, fil is the fault insertion mechanism which inserts faults into the memory image of the task. Compute: is an identifier given to the first fault insertion of the class. compute.attributes refers to the domain extracted from the workload and select_all_by_one refers to the function used to create the fault list from the domain. This function applies the fault mechanism to all members of the domain. The creation of the fault list is controlled by the user after the workload is developed, domains extracted, and fault insertion method(s) defined.

```
#
# Explanatory / Identifying Comments
#

Mechanism: Fault_Injection_Mechanism_Id

Field-1: Domain-1  Method-1
Field-2: Domain-2  Method-2
         .
         .
         .
Field-n: Domain-n  Method-n
```

Figure 14: **Fault Class Format**

```
#
# This is an illustrative example of a Fault Class Definition.
# Comments may be included to describe the Fault Class.
# Generalized Domain Names and Method Names are used.
#

Mechanism: fil

Compute:     compute.attributes      select_all_by_one
Task:        task.attributes         select_all_by_one
.      .       .
 .
```

Figure 15: **Memory Fault Class Example**

As in any abstract type, the fault class can have several instantiations, meaning that each method (e.g., select_all_by_one) can be applied to a different domain and thus generate a specific fault (*fault*

[6]The domain is provided by the task attribute extractor, discussed in Section 5.1.

instance). This process is done automatically in FIAT by a tool named the Fault Instance Generator. Figure 16 depicts the fault instance generation process while Figure 17 shows a list of fault instances.

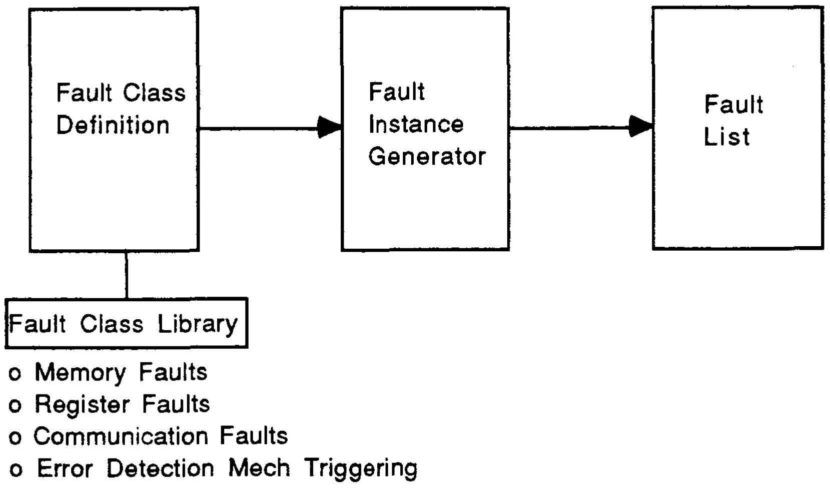

Figure 16: **Fault Instance Generation Process**

Mech	Fire	Sensor	Task	Element	Size	Position	Mask	Behavior
fi1	1	11	mat1	compute	4	0	\c0\00\00\00	xor
fi1	1	11	mat1	compute	4	4	\c0\00\00\00	xor
fi1	1	11	mat1	compute	4	8	\c0\00\00\00	xor
fi1	1	11	mat1	compute	4	12	\c0\00\00\00	xor
fi1	1	11	mat1	compute	4	16	\c0\00\00\00	xor

Figure 17: **Typical Fault List Example**

The columns of the fault instances shown in Figure 17 are: the fault insertion mechanism, `Mech`, defined in the Fault Class; `Fire` is the target hardware described in Section 6.1; `Sensor` is the task observability defined at task generation time (Figure 11); `Task` is the name of the task within the workload; `Element` designates the target domain for the fault insertion; `Size` and `Position` define the size of the faulted word and the location for fault insertion within the element; And `Mask` is a hexadecimal word(s) used to corrupt the target word with the function used in the `Behavior` column.

5.3 Experiments

Experiments are defined by *experiment descriptions* and executed by *experimental scripts*. The *experimental description* is a high level description of experiment flow, which includes workload management, fault insertion, and data collection commands. Figure 18 shows the general format of such a description, while Figure 19 shows an example of an experiment description. Items in the experiment description include: experiment name, database name, workload and fault class, duration of experiment, and the number of runs in the experiment.

```
EXPERIMENT  <experiment_id>
DATABASE  <database_id>
INITFCD     <fcd_id>
LOOP <iteration_cnt>
        WORKLOAD <workload_id>, <run_time>
        FAULT  <fcd_id>, <number_of_faults>
        COLLECT
END
```

Figure 18: **Experiment Description Format**

```
/***
        Experiment Description for Experiment 2
***/

experiment 2

   database matmultiplication.db   #   Define database
   initfcd  direct                 #   Initialize fault class
   loop 100                        #   100 Iterations
      workload matdemo.wl, 30      #   Use matdemo workload
                                   #     30  seconds each
      fault direct, 1              #   Use direct fault class
                                   #     One fault per run
      collect                      #   Collect data after each experiment
   end
```

Figure 19: **Experiment Definition Example**

The experiment description is automatically transformed by the Experiment Description Translator (EDT), into an *experiment script*. The experiment script contains the low level command sequence for controlling the run-time fault insertion. Figure 20 shows a single run of an experiment script generated from the experiment description in Figure 19. Description of the script is given by in-line comments.

```
#
#  Experiment Script for Matrix with Fault Insertion
#
#  Transfer Workload task images to the FIRE machines
#  Workload is matdemo
#
xfer to fire1 matprm1.ti
xfer to fire2 matprm2.ti
xfer to fire1 mat1.ti
xfer to fire2 mat2.ti
xfer to fire1 compare.ti
#
#  Title Experiment
#  SYNTAX : fmc MACHINE echo BEGIN run_id
#  Typical Command sequence begins here
fmc fire1 echo BEGIN 1
fmc fire2 echo BEGIN 1
#
#  Send command to FMC (Fire Monitor and Control)
#  to delay fault insertion until sensor 1 occurs
#  then fault inject matrix a in task Mat1.   See Figure 14.
#  SYNTAX : fmc MACHINE delay SENSOR fii TASKID BEHAV ELEM POS SIZE MASK
#
fmc fire1 delay 26  fii mat1 xor compute 0 4 \c0\00\00\00
#
#  Create Workload on FIRE
#  SYNTAX : fmc MACHINE wcreate task_id image_id
#
fmc fire1 wcreate PRM matprm1
fmc fire2 wcreate PRM matprm2
#
#  Wait for experiment time out
#
pause 30
#
#  Kill Workload
#
fmc fire1 wkill
fmc fire2 wkill
#
#  Collect Data
#  SYNTAX : fmc MACHINE collect RUN_ID DATAFILE [overwrite]
#
fmc fire1 collect 1 xf_1_2_1.dat overwrite
fmc fire2 collect 1 xf_1_2_2.dat overwrite
#
#  Transfer Data to FIM
#
xfer from fire1 xf_1_2_1.dat
xfer from fire2 xf_1_2_2.dat
#
fmc fire1 echo BEGIN 2
```

Figure 20: **Experiment Script Example for One Run**

5.4 Data Collection and Analysis

There are two forms of data collection (DC): histories and error reports. Histories are records of normal functions and performance events. Error reports are records of exceptions and abnormal events. Data analysis supports three hierarchical levels of experimental data processing: run, experiment, and multiple experiments. A run is a single fault insertion in a specific workload. An experiment is a collection of runs, usually from the same fault class. Multiple experiment analysis refers to the ability to correlate a number of experiments under a specific analysis.

In FIAT, two types of data analysis are available: generic and open. The generic data analysis provides a set of predefined functions, such as workload profiling and error coverage statistics. The open data analysis is a relational data base query language which enables the user to define their own analysis. Examples of the data collection/analysis functionality will be presented in Section 7.

6 Implementation

Since the FIAT system is geared toward fault inserting real-time distributed systems, its physical implementation is a real-time distributed system.

6.1 Hardware

Figure 21 presents the hardware structure of the current FIAT system, as implemented at CMU, illustrating the distribution of the abstractions between the two components: the Fault Injection Manager (FIM) and the Fault Injection Receptacles (FIRE). The two parts communicate via a local area network(LAN). The purpose of the FIM is twofold. First, it supports experiment development and data collection/analysis and second, it provides run time control of the experiment. The FIREs provide the execution platform for the distributed system under test. Further descriptions of the FIM and FIRE are in the following section on software.

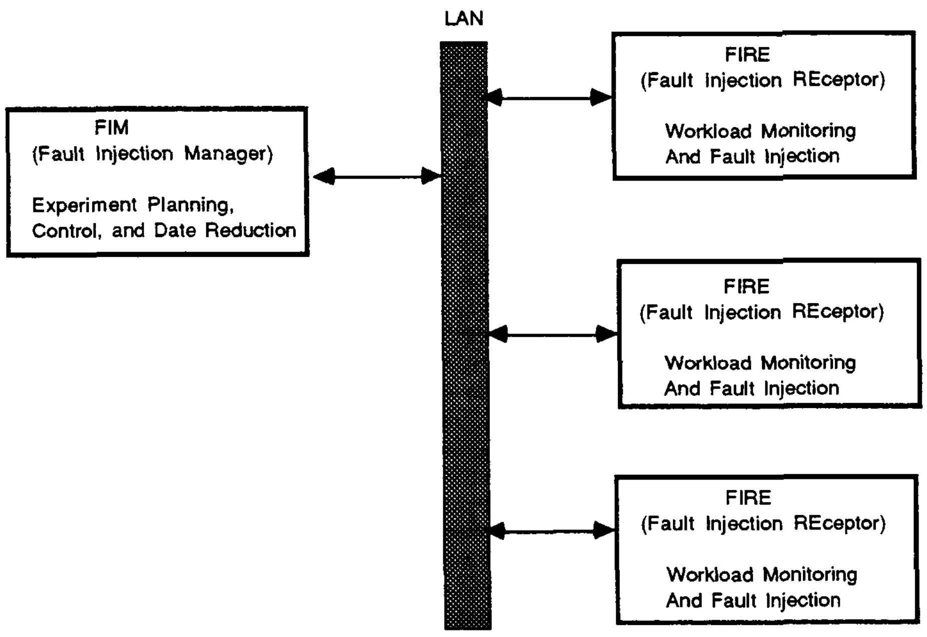
LAN
FIM
(Fault Injection Manager)
Experiment Planning, Control, and Date Reduction
FIRE
(Fault Injection REceptor)
Workload Monitoring And Fault Injection
FIRE
(Fault Injection REceptor)
Workload Monitoring And Fault Injection
FIRE
(Fault Injection REceptor)
Workload Monitoring And Fault Injection

Figure 21: **FIAT Hardware Architecture**

6.2 Software

The FIAT software is similarly divided into two parts: FIM and FIRE as described in the following two subsections.

6.2.1 FIM Software

Figure 22 shows a block diagram of the FIM software. The names of the components of the FIM software are representative of their functionality in the process of generating workloads, fault lists, and experiment scripts, and their control at run time. The Workload Manager, Fault Manager, Experiment Description Manager, Data Analysis Manager, and Data Collection Manager support the respective abstractions at development time, as outlined in the previous sections. The Experiment Manager is responsible for the run time communication with the FIRE machines. The relational database and its software is central to the data storage and analysis functions.

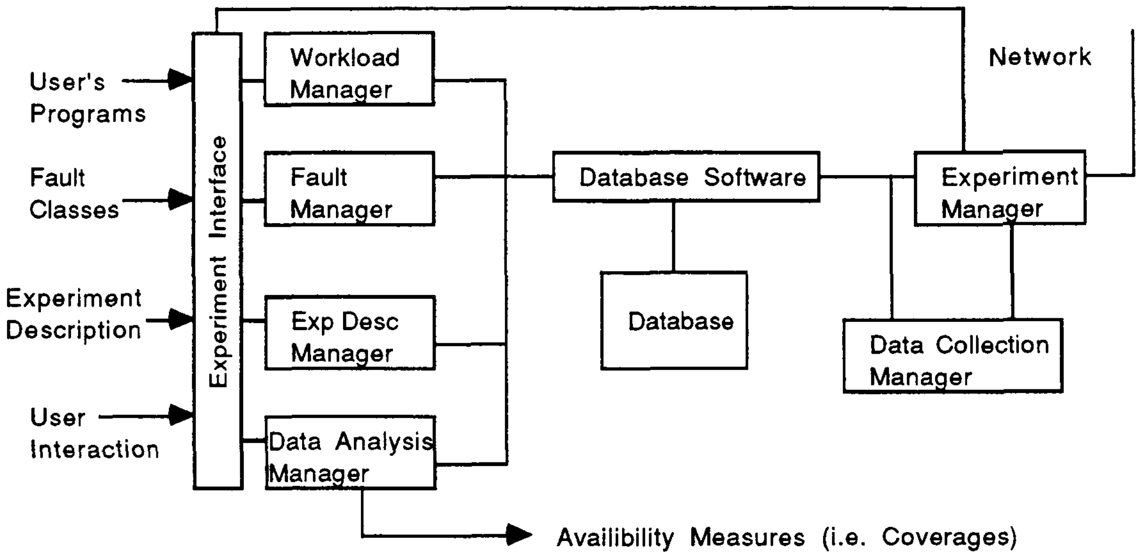

Figure 22: **FIM Software**

The Experiment Interface, depicted in Figure 22, is responsible for the integration of the components into an interactive, purposeful experimentation system. The Menu Manager, in the experiment interface, controls the menus for four modes. Figure 23 shows the menu tree. The four modes involved in the experimentation process include:

- Library Preparation assists preparation of the workload and fault class libraries. The creation of the workload library involves two processes: the task development process and workload definition process. The task development process includes task definition (program creation and editing), task image generation (compilation and linking), and attribute extraction (for use in fault insertion process). The fault class librarian groups the domains and methods used for selecting the fault insertion method during experiment definition.
- Experiment Definition assists in experiment description preparation and experiment script generation through the use of a text editor in creating the description and automatic generation of experiment scripts.

- Experiment Execution provides a list of prepared experiment scripts and executes the selected script.
- Data Analysis provides two types of data analyses: user defined queries using the Informix Structural Query Language (ISQL) and generic FIAT data analysis routines.

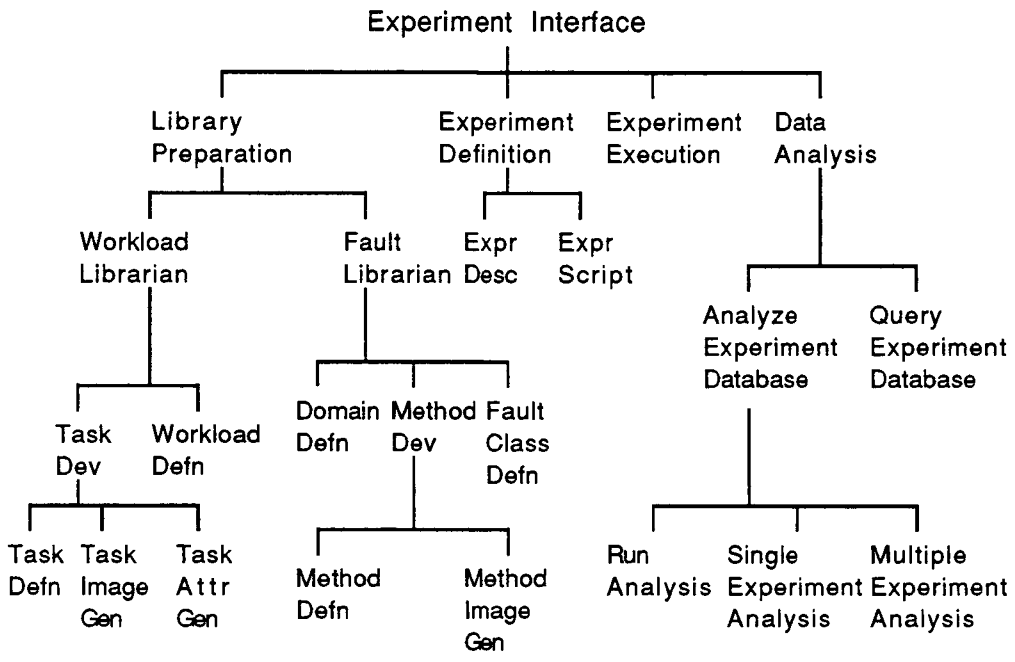

Figure 23: **Experiment Interface Menu Tree**

6.2.2 FIRE Software

The core of the of the FIRE software is the FIRE Monitor and Control (FMC) which is centered around two components depicted in Figure 24: the user defined workload and the modified operating system (OS) kernel. The FMC acts as an executive intermediary between the workload tasks and the operating system providing the monitor and control functions needed for experimentation. The Modified OS Kernel allows the monitoring and fault insertion of the workload tasks. Figure 25 shows a more detailed view of the FIRE software.

There are four modules in the FMC: the Command Controller, the Workload Monitor Controller, the Fault Dispatcher and the Data Collection modules. Their functions are described below.

- Command Controller (CC): Stores and organizes commands from the Experiment Manager into a command queue, and provides synchronization mechanisms to coordinate the commands, such as the starting of tasks and fault insertion within the workload control flow.

It also manages the communication functions required for command transfer and execution. Fault insertion commands are relayed to the Fault Dispatcher for follow-on execution.

- Workload Monitor Controller (WMC): Provides the functionality required for workload monitoring and control, such as task control, communication, and synchronization. Task control includes such commands as task creation, task termination, task suspension and task resumption. Task communication commands include channel creation, send to channel, receive from channel, and select channel. Synchronization commands include event signaling and handling.

- Fault Dispatcher (FD): Manages fault insertion of workload tasks, communication, and the Operating System. The fault insertion commands are transferred to the appropriate entity (e.g., task, OS), where the actual fault insertion is performed. Workload faults include communication and memory faults. OS faults include memory and register faults as well as the triggering of existing hardware and/or software error detection mechanisms.

- Data Collection (DC): Transfers data records generated by the monitor and error report commands, which are user specified, to a local data storage object.

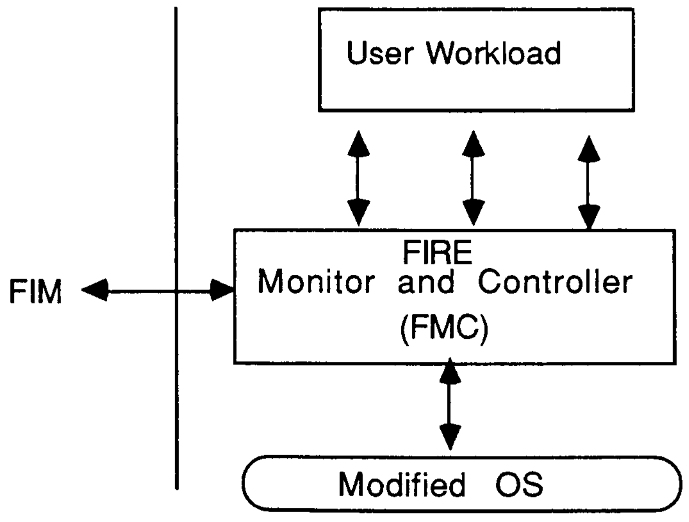

Figure 24: **FIRE Software**

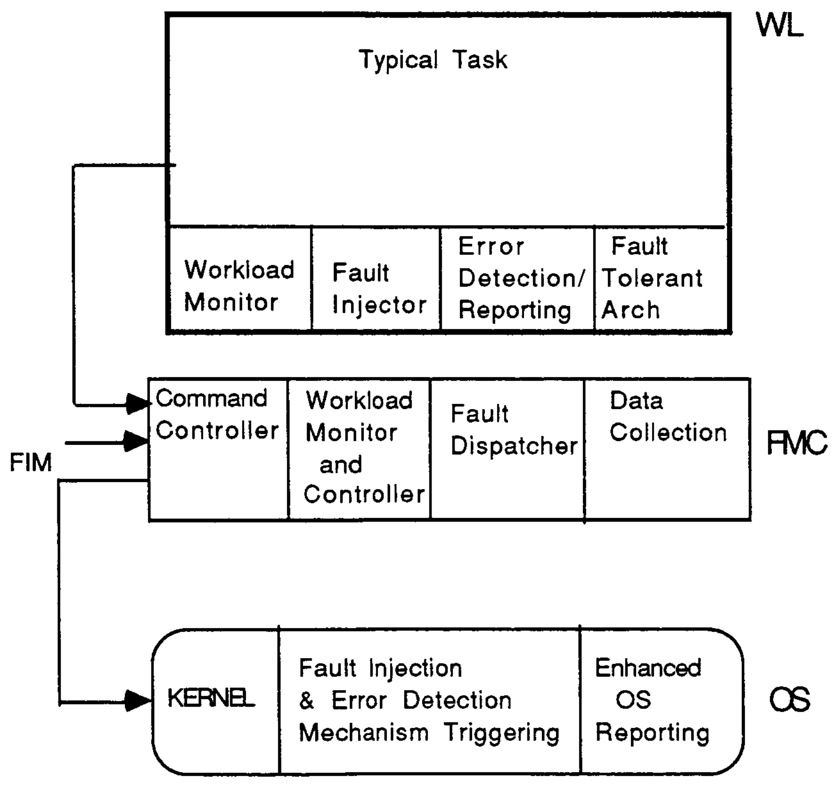

Figure 25: **Detailed view of FIRE Software**

6.3 Experimental Procedure

The experimental procedure ties the various parts of the FIAT system together into a unified environment capable of generating workloads, instrumenting tasks, performing fault-free characterization, designing fault lists, executing fault insertion experiments, and analyzing collected data. Figure 26 describes this process, showing the library preparation of the workload, (Section 5.1), preparation of the fault library, (Section 5.2), the experiment definition (Section 5.3), the experiment execution, and the data analysis (Section 5.4).

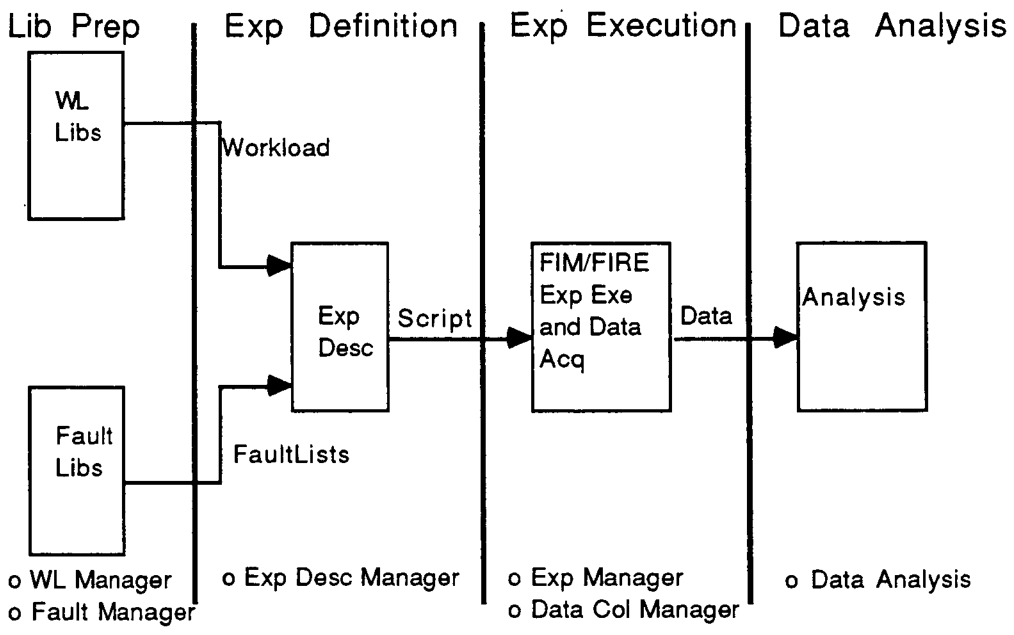

Figure 26: **Typical Fault Insertion Experiment**

7 Evaluation

The FIAT system has been implemented and evaluated since June 1987. The initial results indicate that the system fulfills the stated goals. As evidence of this fulfillment, two experiments performed on the FIAT system are presented in the following subsections.

7.1 Experiment 1

A distributed checkpointing system, utilizing two FIRE machines, was implemented. Figure 27 shows the functional block diagram of the fault tolerance strategy involved. There are two computational engines: the primary and the secondary. The primary, at the start of its computation, informs the secondary of the task, as well as the time frame for the next interaction. The primary then executes the computation and the secondary waits for the next interaction. If the time between interactions exceeds the time frame (i.e., primary failure), the secondary then initiates a recovery action and becomes the primary. If the primary detects that no secondary exists (i.e., secondary failure), it creates a secondary.

The experimental results of fault inserting this system are shown in Figure 28. The fault class was a bus fault consisting of a double bit compensating error. These double errors would not be detected by hardware parity and thus must wait for software error detection. Note the large error latency and that one quarter of the inserted faults were not detected. To illustrate how fault-tolerance strategies may be compared, the above fault-tolerance technique was enhanced by adding checksumming to each computational engine. A checksum is appended to code and data blocks during compilation. The code is regenerated when reading the blocks from memory during runtime and is compared with the predetermined code to detect errors. Hence the detection methods are timeout for the first experiment versus checksums for the second. The improved results are shown in Figure 29. Note the increased fault detection probability and the decreased time to detect.

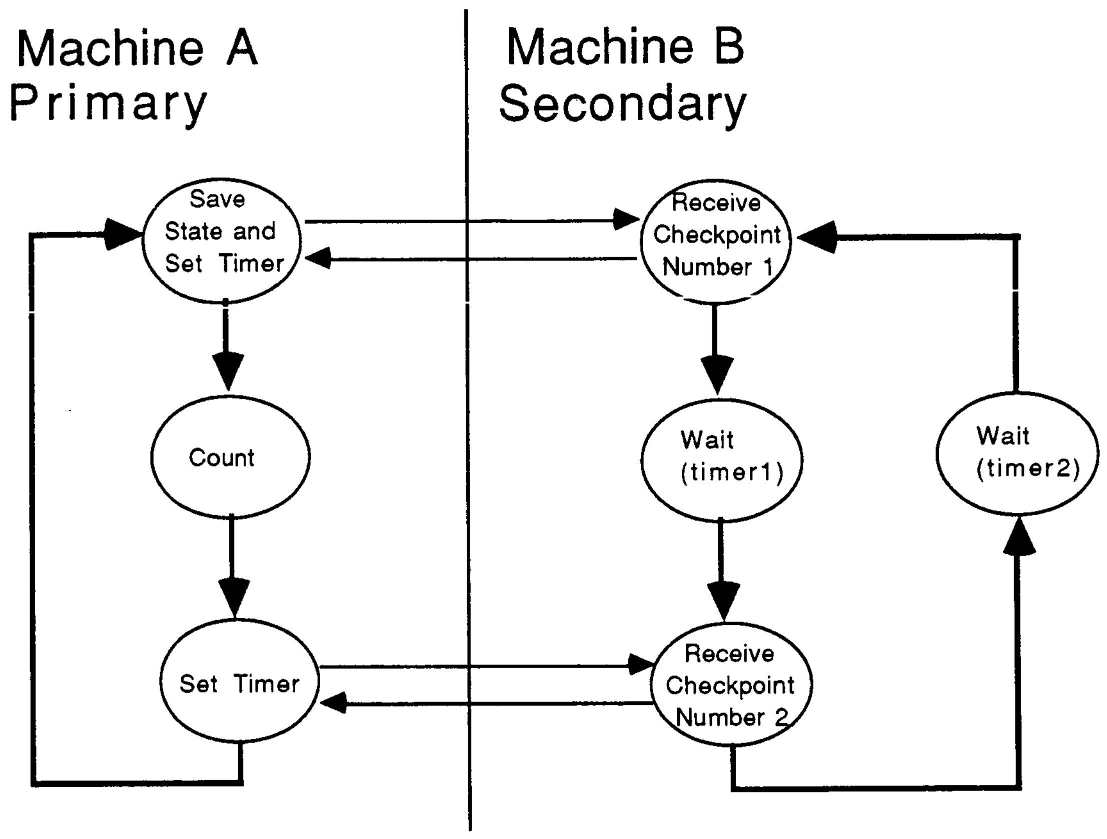

Figure 27: **Real Time Checkpointing Workload**

```
============================================================
METHOD:: Detection Statistics
============================================================

Experiment = 1a
Detection Statistics ...

Number of Fault Injections ... 206
Number of Fault Detections ... 153
Detection Coverage = 74.271845%

Avg Detection Time = 4.219567 seconds
Min Detection Time = 1.406250 seconds
Max Detection Time = 6.218750 seconds
```

Figure 28: **Experiment 1: Error Detection Statistics (Checkpointing Only)**

```
==========================================================
METHOD:: Detection Statistics
==========================================================

Experiment = 1b
Detection Statistics ...

Number of Fault Injections ... 91
Number of Fault Detections ... 86
Detection Coverage = 94.505495%

Avg Detection Time = 3.102834 seconds
Min Detection Time = 1.203125 seconds
Max Detection Time = 4.203125 seconds

END OF METHOD: Detection Statistics
```

Figure 29: **Experiment 1: Error Detection Statistics (Checkpointing with Checksums)**

7.2 Experiment 2

A software duplicate and match fault tolerant strategy was implemented. The outputs of two identical computational engines, located in different FIREs and performing the same computation (matrix multiplication), are compared on a third computational engine. Experimentation consisted of fault inserting one of the identical engines. The experimental results of fault inserting with two different fault classes, a bus fault and a memory fault, are shown in Figure 30 and Figure 31. Note that while the error detection coverage varies between experiments (63.0% vs. 54.5%), there is almost an identical distribution of fault manifestations (e.g. Stop, Invalid Output, Response Too Late, Crash and Hung).

```
Netmat3 Experiment Set 1 - Two-bit Compensating Errors
440 Faults Injected Per Experiment: 8800 inserted faults

Exp ID    Faults Detected    % Detected      Mask: XOR

   1           287             65.2         \c0\00\00\00
   2           274             62.3         \08\02\00\00
   3           270             61.4         \00\44\00\00
   4           289             65.7         \00\01\01\00
   5           255             58.0         \00\00\00\a0
   6           279             63.4         \10\00\00\01
   7           239             54.3         \0a\00\00\00
   8           250             56.8         \00\0a\00\00
   9           242             55.0         \00\00\0a\00
  10           279             63.4         \00\00\80\02
  11           279             63.4         \00\02\00\40
  12           298             67.7         \00\20\00\20
  13           289             65.7         \00\08\00\04
  14           297             67.5         \20\00\40\00
  15           305             69.3         \01\00\00\08
  16           297             67.5         \01\00\00\40
  17           259             58.9         \00\00\03\00
  18           277             63.0         \04\00\80\00
  19           302             68.6         \02\00\00\04
  20           278             63.2         \00\02\00\01

Mean:  277.25 detections (63.0% coverage)
Standard Deviation:  19.62 (7.08% of the mean)

Error Detection Mechanisms

Mechanism                    # Detected      % of Total

Abnormal Task Death (STOP)     3 124            56.3
Invalid Output                 1 460            26.3
Response Too Late                879            15.9
Machine Reboot (CRASH)            80             1.4
Processor Halt (HUNG)              2             0.0 *

Totals     ======>             5 545           100.0
```

Figure 30: **Error Detection Statistics: Two Bit Compensating Errors**

```
Netmat3 Experiment Set 2 - 'Zero-a-byte' Errors
440 Faults Injected per Experiment: 8800 inserted faults

Exp ID    Faults Detected     % Detected      Mask: AND

   1            258             58.6         \00\ff\ff\ff
   2            229             52.0         \ff\00\ff\ff
   3            240             54.5         \ff\ff\00\ff
   4            217             49.3         \ff\ff\ff\00
   5            239             54.3         \f0\0f\ff\ff
   6            260             59.1         \ff\f0\0f\ff
   7            227             51.6         \ff\ff\f0\0f
   8            244             55.5         \c0\3f\ff\ff
   9            236             53.6         \ff\c0\3f\ff
  10            239             54.3         \ff\ff\c0\3f
  11            258             58.6         \80\7f\ff\ff
  12            213             48.4         \ff\80\7f\ff
  13            243             55.2         \ff\ff\80\7f
  14            245             55.7         \e0\1f\ff\ff
  15            235             53.4         \ff\e0\1f\ff
  16            235             53.4         \ff\ff\e0\1f
  17            249             56.6         \ff\ff\fe\01
  18            219             49.8         \ff\ff\fc\03
  19            260             59.1         \ff\f8\07\ff
  20            249             56.6         \ff\fe\01\ff

Mean:  239.75 detections (54.5% coverage)
Standard Deviation:  13.98 (5.83% of the mean)


Error Detection Mechanisms

Mechanism                     # Detected      % of Total

Abnormal Task Death (STOP)    2 748             57.3
Invalid Output                1 249             26.0
Response Too Late               753             15.7
Machine Reboot (CRASH)           45              0.9
Processor Halt (HUNG)             0              0.0

Totals     ======>            4 795            100.0
```

Figure 31: Error Detection Statistics: Zero-a-byte Errors

8 Summary

The goals, concepts, design, implementation, and evaluation of a Fault Injection-based Automated Testing environment have been presented. The initial evaluation has shown two things. First, the initial goals have been met. Second, the evaluation process, applied to two experiments, has shown how measurements can give hints at improving the architecture under evaluation. Currently, the main thrusts of the project are in the following areas:

- Reducing of the complexity of fault insertion on all levels.
- Correlating the fault insertion manifestations to realistic software errors.
- Providing absolute measures of real-time distributed system dependability.
- Applying FIAT in the design process of realistic systems.
- Refining the process of predeployment validation using FIAT.

To the authors, a tool is as relevant as its applicability to either acquiring experimental evidence for a new theory or applying it to solve real life problems. The FIAT system is a step toward fulfilling these desiderata. The example fault-tolerant strategies evaluated in FIAT provided initial proof of the system usefulness in both areas.

9 References

[1] Jacob A. Abraham and W. Kent Fuchs. Fault and error models for VLSI. *Proceedings of the IEEE*, 74(5):639–654, May 1986.

[2] Allen T. Acree, Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. *Mutation Analysis.* Technical Report GIT-ICS 79/08, Georgia Institute of Technology, August 1982.

[3] Allen Troy Acree, Jr. *On Mutation.* PhD thesis, Georgia Institute of Technology, School of Information and Computer Science, August 1980.

[4] W. Richards Adrion, Martha A. Branstad, and John C. Cheriavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, June 1982.

[5] Algirdas Avižienis and Jean-Cluade Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986.

[6] Algirdas Avižienis and David A. Rennels. Fault-tolerance experiments with the JPL Star computer. In *Digest of Papers, COMPCON '72*, pages 321–324, September 1972.

[7] C.C. Beh, K.H. Arya, C.E. Radke, and K.E. Torku. Do stuck faults reflect manufacturing defects? In *1982 IEEE Test Conference*, pages 35–42, 1982.

[8] L.A. Boone, H.L. Liebergot, and R.M. Sedmak. Availability, reliability, and maintainability aspects of the Sperry UNIVAC 1100/60. In *10th International Symposium on Fault-Tolerant Computing*, pages 3–9, 1980.

[9] R.E. Bryant. A switch level model and simulator for MOS digital circuits. *IEEE Transactions on Computers*, C-33(2):160–177, February 1984.

[10] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. *Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs.* Technical Report GIT-ICS 80/01, Georgia Institute of Technology, February 1980.

[11] W.C. Carter. System validation - Putting the pieces together. In *7th AIAA/IEEE Digital Avionics Systems Conference*, pages 687–694, 1986.

[12] Ed Clune, Zary Segall, and Daniel Siewiorek. *Fault-Free Behavior of Reliable Multiprocessor Systems: FTMP Experiments in AIRLAB.* NASA CR-177967, Carnegie Mellon Univ., August 1985.

[13] Bernard Courtois. Some results about the efficiency of simple mechanisms for the detection of microcomputer malfunctions. In *9th International Symposium on Fault-Tolerant Computing*, pages 71–74, 1979.

[14] Y. Crouzet and B. Decouty. Measurement of fault detection mechanisms efficiency: Results. In *12th International Symposium on Fault-Tolerant Computing*, pages 373–376, 1982.

[15] Edward W. Czeck, Frank E. Feather, Ann Marie Grizzaffi, Zary Z. Segall, and Daniel P. Siewiorek. *Fault-Free Performance Validation of Fault-Tolerant Multiprocessors.* NASA CR-178236, Carnegie Mellon Univ., January 1987.

[16] Edward W. Czeck, Daniel P. Siewiorek, and Zary Z. Segall. *Software Implemented Fault Insertion: An FTMP Example.* NASA CR-178423, Carnegie Mellon Univ., October 1987.

[17] Scott Davidson. Fault simulation at the architectural level. In *International Test Conference*, pages 669–679, 1984.

[18] Scott Davidson and James L. Lewandowski. ESIM/AFS - A concurrent architectural level fault simulator. In *International Test Conference*, pages 375–383, 1986.

[19] B. Decouty, G. Michel, and C. Wagner. An evaluation tool of fault detection mechanisms efficiency. In *10th International Symposium on Fault-Tolerant Computing*, pages 225–227, 1980.

[20] Yves Deswarte, Khadija Alami, and Oliver Tedaldi. Realization, validation and operation of a fault-tolerant multiprocessor: ARMURE. In *16th International Symposium on Fault-Tolerant Computing*, pages 8–13, 1986.

[21] Fausto Fantini. Reliability problems with VLSI. *Microelectronics Reliability*, 24(2):275–296, 1984.

[22] Frank Feather, Daniel Siewiorek, and Zary Segall. *Fault-Free Validation of a Fault-Tolerant Multiprocessor: Baseline Experiments and Workload Implementation.* NASA CR-178075, Carnegie Mellon Univ., April 1986.

[23] F. Joel Ferguson. *Inductive Fault Analysis of VLSI Circuits.* PhD thesis, Carnegie Mellon University, Electrical and Computer Engineering Dept., October 1987.

[24] George B. Finelli. Characterization of fault recovery through fault injection on FTMP. *IEEE Transactions on Reliability*, R-36(2):164–170, June 1987.

[25] J. Galiay, Y. Crouzet, and M. Vergniault. Physical versus logical fault models MOS LSI circuits: Impact on their testability. *IEEE Transactions on Computers*, C-29(6):527–531, June 1980.

[26] Frank M. Goetz. Design for detection, an attempt at complete fault detection of a store. In *Digest of Papers, COMPCON '72*, pages 325–328, September 1972.

[27] Gary L. Hartmann, Joseph E. Wall, Jr., and Edward R. Rang. Design validation of fly-by-wire flight control systems. In *AGARD Lecture Series No. 143, Fault Tolerant Hardware/Software Architecture for Flight Critical Function*, pages 9.1–9.17. NATO Advisory Group for Aerospace Research and Development, 1985.

[28] John P. Hayes. Fault modeling. *IEEE Design and Test*, pages 88–95, April 1985.

[29] H.M. Holt, A.O. Lupton, and D.G. Holden. Flight critical system design guidelines and validation methods. In *AIAA/AHS/ASEE Aircraft Design Systems and Operating Meeting*, 1984. Paper: AIAA-84-2461.

[30] William E. Howden. Functional program testing. *IEEE Transactions on Software Engineering*, SE-6(2):162–169, March 1980.

[31] Raymond P. Kurlak and J. Robert Chobot. CPU coverage evaluation using automatic fault injection. In *4th AIAA/IEEE Digital Avionics Systems Conference*, pages 294–300, 1981. 81-2281.

[32] Kwok-Woon Lai. *Functional Testing of Digital Systems.* PhD thesis, Carnegie-Mellon University, Dept. of Computer Science, December 1981. Technical report CMU-CS-81-148a.

[33] Kwok-Woon Lai and Daniel P. Siewiorek. Functional testing of digital systems. In *20th Design Automation Conference*, pages 207–213. IEEE, 1983.

[34] Larry Kwok-Woon Lai. Error-oriented architecture testing. In *National Computer Conference*, pages 565–576, June 1979.

[35] Jaynarayan H Lala and T. Basil Smith III. *Development and Evaluation of a Fault-Tolerant Multiprocessor Computer, Vol. III, FTMP Test and Evaluation.* NASA CR-166073, Charles Stark Draper Laboratories, May 1983.

[36] J.H. Lala. Fault detection isolation and reconfiguration in FTMP: Methods and experimental results. In *5th AIAA/IEEE Digital Avionics Systems Conference*, pages 21.3.1–21.3.9, 1983.

[37] Jean-Cluade Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *15th International Symposium on Fault-Tolerant Computing*, pages 2–11, 1985.

[38] J.R. Lloyd and J.A. Knight. The relationship between electromigration-induced short-circuit and open-circuit failure times in multi-layer VLSI technologies. In *International Reliability Physics Symposium*, pages 48–51, 1984.

[39] Tülin Erdim Mangir. Sources of failures and yield improvement for VLSI and restructurable interconnects for RVLSI and WSI: Part I sources of failure and yield improvement for VLSI. *Proceedings of the IEEE*, 72(6):690–708, June 1984.

[40] P. Marchal. Updating functional fault models for microprocessors internal buses. In *15th International Symposium on Fault-Tolerant Computing*, pages 58–64, 1985.

[41] John G. McGough and Fred Swern. *Measurement of Fault Latency in a Digital Avionic Mini Processor.* NASA CR-3462, Bendix Corp., October 1981.

[42] John G. McGough and Fred Swern. *Measurement of Fault Latency in a Digital Avionic Mini Processor, Part II.* NASA CR-3651, Bendix Corp., January 1983.

[43] John G. McGough, Fred Swern, and Salvatore J. Bavuso. Methodology for measurement of fault latency in a digital avionic miniprocessor. In *4th AIAA/IEEE Digital Avionics Systems Conference*, pages 310–314, 1981. 81-2282.

[44] John G. McGough, Fred Swern, and Salvatore J. Bavuso. New results in fault latency modeling. In *Proceeding of the IEEE EASCON Conference*, pages 299–306, August 1983.

[45] P. Michael Melliar-Smith and Richard L. Schwartz. Formal specification and mechanical verification of SIFT: A fault-tolerant flight control system. *IEEE Transactions on Computers*, C-31(7):616–630, June 1982.

[46] Matthew J. Middendorf and Tom Hausken. Observed physical effects and failure analysis of EOS/ESD on MOS devices. In *International Symposium for Testing and Failure Analysis*, pages 205–213, October 1984.

[47] *Validation Methods for Fault-Tolerant Avionics and Control Systems: Working Group Meeting I*, NASA Langley Research Center, March 1979. ORI, Incorporated, Compilers. NASA CP-2114.

[48] *Validation Methods for Fault-Tolerant Avionics and Control Systems: Working Group Meeting II*, NASA Langley Research Center, September 1979. System and Measurements Division, Research Triangle Institute. NASA CP-2130.

[49] J. Duane Northcutt. The design and implementation of fault insertion capabilities for ISPS. In *18th Design Automation Conference*, pages 197–209. IEEE, 1980.

[50] Daniel L. Palumbo and George B. Finelli. *A Technique for Evaluating the Application of the Pin-Level Stuck-at Fault Model to VLSI Circuits.* NASA TP-2738, Langley Research Center, September 1987.

[51] SAE Committee S-18A. *Fault/Failure Analysis for Digital Systems and Equipment.* Aerospace Recommended Practice ARP-1834, Society of Automotive Engineers, Warrendale, Pa., August 1986.

[52] M.E. Schmid, R.L. Trapp, A.E. Davidoff, and G.M. Masson. Upset exposure by means of abstract verification. In *12th International Symposium on Fault-Tolerant Computing*, pages 237–244, 1982.

[53] M.A. Schuette, J.P. Shen, D.P. Siewiorek, and Y.X. Zhu. Experimental evaluation of two concurrent error detection schemes. In *16th International Symposium on Fault-Tolerant Computing*, pages 138–143, 1986.

[54] John P. Shen, W. Maly, and F. Joel Ferguson. Inductive fault analysis of MOS integrated circuits. *IEEE Design and Test of Computers*, 2(6):13–26, December 1985.

[55] Kang G. Shin and Yann-Hang Lee. Measurement and application of fault latency. *IEEE Transactions on Computers*, C-35(4):370–375, April 1986.

[56] Daniel P. Siewiorek and Robert S. Swarz. *The Theory and Practice of Reliable System Design.* Digital Press, Bedford MA, 1982.

[57] Gabriel M. Silberman and Ilan Spillinger. The difference fault model using functional fault simulation to obtain implementation fault coverage. In *International Test Conference*, pages 332–339, 1986.

[58] J.J. Stiffler and A.H. Van Doren. FTSC - Fault tolerant spaceborne computer. In *9th International Symposium on Fault-Tolerant Computing*, page 143, 1979.

[59] Stephen Y.H. Su and Tonysheng Lin. Functional testing techniques for digital LSI/VLSI devices. In *21st Design Automation Conference*, pages 517–528. IEEE, 1984.

[60] S. Thatte and J. Abraham. A methodology for functional level testing of microprocessors. In *8th International Symposium on Fault-Tolerant Computing*, pages 90–95, 1978.

[61] S. Thatte and J. Abraham. Test generation for general microprocessor architectures. In *9th International Symposium on Fault-Tolerant Computing*, pages 203–210, 1979.

[62] Satish M. Thatte and Jacob A. Abraham. Test generation for microprocessor. *IEEE Transactions on Computers*, C-29(6):429–441, June 1980.

[63] C. Timoc, M. Buehler, T. Griswold, C. Pina, F. Scott, and L. Hess. Logical models of physical failures. In *International Test Conference*, pages 546–553, 1983.

[64] R.L Wadsack. Fault modeling and logic simulation of CMOS and MOS integrated circuits. *Bell Systems Technical Journal*, 57(5):1449–1473, May-June 1978.

[65] X.Z. Yang, G. York, W.P. Birmingham, and D.P. Siewiorek. Fault recovery of triplicated software on the Intel iAPX 432. In *Distributed Computing Systems*, pages 438–443, May 1985.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-4244	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Predeployment Validation of Fault-Tolerant Systems Through Software-Implemented Fault Insertion		5. Report Date July 1989
		6. Performing Organization Code
7. Author(s) Edward W. Czeck, Daniel P. Siewiorek, and Zary Z. Segall		8. Performing Organization Report No.
		10. Work Unit No. 505-66-21-01
9. Performing Organization Name and Address Carnegie-Mellon University Electrical and Computer Engineering Department Schenley Park Pittsburgh, PA 15213		11. Contract or Grant No. NAG1-190
		13. Type of Report and Period Covered Contractor Report 11/87-11/88
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Peter A. Padilla
Final Report

16. Abstract

This report describes FIAT, Fault Injection-based Automated Testing environment, which can be used to experimentally characterize and evaluate distributed realtime systems under fault-free and faulted conditions. The report begins with a survey of validation methodologies, demonstrates the need for fault insertion based on validation methodologies, overviews the origins and models of faults, and provides motivation for the FIAT concept.

FIAT employs a validation methodology which builds confidence in the system through first providing a baseline of fault-free performance data and then characterizing the behavior of the system with faults present. Fault insertion is accomplished through software and allows faults or the manifestation of faults to be inserted by either "seeding" faults into memory or triggering error detection mechanisms.

FIAT is capable of emulating a variety of fault-tolerant strategies and architectures, can monitor system activity, and can automatically orchestrate experiments involving insertion of faults. There is a common system interface which allows ease of use to decrease experiment development and run time. Fault models chosen for experiments on FIAT have generated system responses which parallel those observed in real systems under faulty conditions. These capabilities are shown by two example experiments each using a different fault-tolerance strategy.

17. Key Words (Suggested by Author(s)) Validation, Fault Injection, Fault Tolerance, Software-Implemented Fault Insertion, Distributed Systems		18. Distribution Statement Unclassified - Unlimited; Star Category 62	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 48	22. Price A03

NASA FORM 1626 OCT 86 NASA-Langley, 1989

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171